Contents

HW__Motion__Planning__For__Mobile__Robots

```
for (auto &curr_node : neighbour_nodes)
{
  if (!map_ptr_->isSegmentValid(curr_node->x, x_new))
    continue;
  double temp_dist = curr_node->cost_from_start + calDist(curr_node->x, x_new);
  if (temp_dist < min_dist_from_start)
  {
    min_node = curr_node;
    min_dist_from_start = temp_dist;
    cost_from_p = calDist(curr_node->x, x_new);
  }
}
```

Every time we find a new node, we check all the nodes in the vincinity, and check whether this neighbor node would give us a lower cost from start than the current neighbor. If this neighbor node has a lower cost, we compute update this new node's parent to this neighbor node and update the costs accordingly.

```
for (auto &curr_node : neighbour_nodes)
{
  double best_cost_before_rewire = goal_node_->cost_from_start;
  // ! -----------------------------------
  if (!map_ptr_->isSegmentValid(curr_node->x, x_new))
    continue;
  double temp_dist = new_node->cost_from_start + calDist(new_node->x, curr_node->x);
  if (temp_dist < curr_node->cost_from_start)
    changeNodeParent(curr_node, new_node, calDist(new_node->x, curr_node->x));

  // ! -----------------------------------
  if (best_cost_before_rewire > goal_node_->cost_from_start)
  {
    vector<Eigen::Vector3d> curr_best_path;
    fillPath(goal_node_, curr_best_path);
    path_list_.emplace_back(curr_best_path);
    solution_cost_time_pair_list_.emplace_back(goal_node_->cost_from_start, (ros::Time::now() - rrt_start_time).
  }
}
```

Similar to above, we check the neighbor nodes around the new node. Instead of checking the new node, we check if any of the neighbor nodes can have a lower cost if the neighbor node's parent is the new node. If so, we update this neighbor node's parent to be the new node.

Uniform Sampling for Informed RRT*

```
void samplingOnce(Eigen::Vector3d &sample, bool ball)
  {
    if (ball)
    {
      // Uniformly Sample in Spherical Coord
      double theta = uniform_rand_pi(gen_);
      double phi = uniform_rand_2pi(gen_);
      double radius = std::pow(uniform_rand_(gen_), 1/3.);
      sample[0] = radius * sin(theta) * cos(phi);
      sample[1] = radius * sin(theta) * sin(phi);
      sample[2] = radius * cos(theta);
    }
```

```
    else
    {
      sample[0] = uniform_rand_(gen_);
      sample[1] = uniform_rand_(gen_);
      sample[2] = uniform_rand_(gen_);
      sample.array() *= range_.array();
      sample += origin_;
    }
  };
```

I changed the SamplingOnce method. The boolean ball is true when goal_found == true. Instead of rejection method, I have decided to uniformly sample in the spherical coordinates. Then convert it back to Cartesian Coordinates.

Other code for Informed RRT*

```
Eigen::Matrix3d getRotationMatrix(Eigen::Vector3d v1, Eigen::Vector3d v2)
{
  // Use the unit vectors to find the rotation matrix
  // Find the unit vector of both vector
  v1 /= v1.norm();
  v2 /= v2.norm();
  Eigen::Vector3d v = v1.cross(v2);
  double s = v.norm();
  double c = v1.dot(v2);
  Eigen::Matrix3d skew_sym;
  skew_sym << 0, -v(2), v(1),
              v(2), 0, -v(0),
              -v(1), v(0), 0;
  Eigen::Matrix3d rotation_matrix = Eigen::Matrix3d::Zero();
  rotation_matrix = Eigen::Matrix3d::Identity() + skew_sym + skew_sym * skew_sym * (1 - c) / s / s;
  return rotation_matrix;
}
```

Here is the implementation to find the rotation matrix that rotate v1 (x axis) to v2 (main axis of the elipse). If I remember this correctly, this is just Rodrigue's formula in a different form.

```
Eigen::Vector3d center = (s + g) / 2;
Eigen::Vector3d origin_vec(1, 0, 0);
Eigen::Matrix3d R_mat = getRotationMatrix(origin_vec, g - s);
Eigen::Matrix3d S_mat = Eigen::Matrix3d::Zero();
double c_min = (g - s).norm();
```

The code above are computed prior to the main loop, since they are constant for each search.

```
sampler_.samplingOnce(x_rand, goal_found);
// samplingOnce(x_rand);
if (goal_found)
{
  double c_max = goal_node_->cost_from_start;
  S_mat(0,0) = c_max;
  S_mat(1,1) = sqrt(c_max*c_max - c_min*c_min) / 2;
  S_mat(2,2) = S_mat(1,1);
  x_rand = R_mat * S_mat * x_rand + center;
}
```

Construct the scaling matrix to form the elipse. Formulas taken from Informed RRT* Paper (https://arxiv.org/abs/1404.2334).